

数学计算器项目报告

1. 程序功能简要说明

本项目是一个基于 Tauri 框架的桌面数学计算器应用程序，集成了两个主要功能模块：

1.1 数学表达式计算器

- **功能**：支持复杂数学表达式的计算，包括加减乘除、乘方、绝对值等运算
- **特点**：采用栈式算法实现，支持运算符优先级和括号匹配
- **可视化**：提供栈操作动画，实时展示计算过程中的栈状态变化

1.2 多项式计算器

- **功能**：支持多项式的创建、运算和求导
- **支持运算**：多项式加减乘、求导、表达式求值
- **存储管理**：支持多个多项式 (a-e) 的存储和管理，如有需要，可以在代码扩充多项式数量
- **显示**：支持 LaTeX 格式的多项式显示

1.3 技术架构

- **前端**：React 19 + TypeScript + Vite
- **后端**：Rust + Tauri 框架
- **计算引擎**：C++ 实现的高性能栈式算法

2. 功能展示

最佳展示效果请观看demo.mp4或直接操作.exe应用

算术表达式计算及栈动画展示1



多项式建立展示1

cpptest

一元稀疏多项式计算器

构建多项式

输入格式: 系数1,指数1,系数2,指数2,... (例如: 3,2,2,1,1,0 表示 $3x^2+2x+1$)
无需项数n, 无需保证指数递减排列

多项式 a ▾

3,2,2,1,1,0

构建

多项式运算

支持运算: + (加法), - (减法), * (乘法), 括号 (例如: a+b, a-b*c, (a+b)*c)

例如: a+b*c

计算

多项式求值与求导

多项式 a ▾

1

求值

求导

清空所有

已创建的多项式

多项式 a: 3,3,2,2,1,1,0

显示

LaTeX

隐藏

结果

标准格式:
3, 3, 2, 2, 1, 1, 0

LaTeX 格式:
$$3x^2 + 2x + 1$$

4 / 19

多项式建立展示2

cpptest

一元稀疏多项式计算器

构建多项式

输入格式: 系数1,指数1,系数2,指数2,... (例如: 3,2,2,1,1,0 表示 $3x^2+2x+1$)
无需项数n, 无需保证指数递减排列

多项式 b ▾

5,2,1,1,1,0

构建

多项式运算

支持运算: + (加法), - (减法), * (乘法), 括号 (例如: a+b, a-b*c, (a+b)*c)

例如: a+b*c

计算

多项式求值与求导

多项式 b ▾

1

求值

求导

清空所有

已创建的多项式

隐藏

多项式 a: 3,3,2,2,1,1,0

显示

LaTeX

多项式 b: 3,5,2,1,1,1,0

显示

LaTeX

结果

标准格式:

3, 5, 2, 1, 1, 1, 0

LaTeX 格式:

$$5x^2 + x + 1$$

多项式表达式计算展示

cpptest

输入格式: 系数1,指数1,系数2,指数2,... (例如: 3,2,2,1,1,0 表示 $3x^2+2x+1$)

无需项数n, 无需保证指数递减排列

多项式 c ▾

5,2,3,1,1,0

构建

多项式运算

支持运算: + (加法), - (减法), * (乘法), 括号 (例如: a+b, a-b*c, (a+b)*c)

(a+b)*c

计算

多项式求值与求导

多项式 c ▾

1

求值

求导

清空所有

已创建的多项式

隐藏

多项式 a: 3,3,2,2,1,1,0

显示

LaTeX

多项式 b: 3,5,2,1,1,1,0

显示

LaTeX

多项式 c: 3,5,2,3,1,1,0

显示

LaTeX

结果

标准格式:

5, 40, 4, 39, 3, 27, 2, 9, 1, 2, 0

LaTeX 格式:

$$40x^4 + 39x^3 + 27x^2 + 9x + 2$$

多项式求值展示

cpptest

多项式求值与求导

多项式 c ▾

1

求值

求导

清空所有

已创建的多项式

隐藏

多项式 a: 3,3,2,2,1,1,0

显示

LaTeX

多项式 b: 3,5,2,1,1,1,0

显示

LaTeX

多项式 c: 3,5,2,3,1,1,0

显示

LaTeX

结果

标准格式:

5, 40, 4, 39, 3, 27, 2, 9, 1, 2, 0

LaTeX 格式:

$$40x^4 + 39x^3 + 27x^2 + 9x + 2$$

求值结果

9

使用示例:

• 构建多项式 a: 输入 "3,2,2,1,1,0" ($3x^2+2x+1$)

• 构建多项式 b: 输入 "1,1,-1,0" ($x-1$)

• 计算 a+b: 输入 "a+b"

• 计算 a*b: 输入 "a*b"

• 多项式求值: 选择多项式 a, 输入 x=2

• 多项式求导: 选择多项式 a, 点击"求导"按钮

多项式求导展示

cpptest

多项式求值与求导

多项式 a

1

求值

求导

清空所有

已创建的多项式

隐藏

多项式 a: 3,3,2,2,1,1,0

显示

LaTeX

多项式 b: 3,5,2,1,1,1,0

显示

LaTeX

多项式 c: 3,5,2,3,1,1,0

显示

LaTeX

结果

标准格式:

2, 6, 1, 2, 0

LaTeX 格式:

$6x + 2$

求值结果

9

使用示例:

构建多项式 a: 输入 "3,2,2,1,1,0" ($3x^2+2x+1$)

构建多项式 b: 输入 "1,1,-1,0" ($x-1$)

计算 a+b: 输入 "a+b"

计算 a*b: 输入 "a*b"

多项式求值: 选择多项式 a, 输入 x=2

多项式求导: 选择多项式 a, 点击"求导"按钮

3. 部分关键代码及其说明

3.1 表达式计算核心算法 (C++)

```
// stack.hpp - 栈使用顺序表实现，类似vector的动态扩容机制
template <typename T>
class Stack {
private:
    T *data_;
    int cnt_;
    size_t capacity_;

public:
    explicit Stack(size_t capacity = 10000)
        : capacity_(capacity), cnt_(0){
        data_ = new T[capacity_];
    }

    ~Stack(){
        delete[] data_;
    }

    void push(const T& item) {
        if (cnt_ == capacity_) {
            capacity_ <<= 1;
            T *tmp = new T[capacity_];
            for (int i = 0; i < cnt_; ++i) {
                tmp[i] = data_[i];
            }
            delete[] data_;
            data_ = tmp;
        }
        data_[cnt_++] = item;
    }

    // ...clear(),top()
}
```

```
// calc_expression.cpp - 双栈算法实现
```

```
//优先级判断函数
```

```
static bool should_operator_execute(char stack_top, char current_input) {
    return (stack_top == '+' && (current_input == '-' || current_input == '+' ||
current_input == ')') || current_input == '|') ||
        (stack_top == '-' && (current_input == '-' || current_input == '+' ||
current_input == ')') || current_input == '|') ||
        (stack_top == '*' && (current_input == '/' || current_input == '*' ||
current_input == '+' || current_input == '-' || current_input == ')') ||
```

```

current_input == '|')) ||
    (stack_top == '/' && (current_input == '/' || current_input == '*' ||
current_input == '+' || current_input == '-' || current_input == ')') ||
current_input == '|')) ||
    (stack_top == '^' && (current_input == '+' || current_input == '-' ||
current_input == '*' || current_input == '/' || current_input == '^' ||
current_input == ')') || current_input == '|')) ||
    (stack_top == '(' && current_input == ')') ||
    (stack_top == '|' && current_input == '|');
}

static pair<int, string> perform_calculation(char operation, int operand_a, int
operand_b) {
    switch (operation) {
        case '+':
            return make_pair(operand_b + operand_a, "");
        case '-':
            return make_pair(operand_b - operand_a, "");
        case '*':
            return make_pair(operand_b * operand_a, "");
        case '/':
            if (operand_a == 0) { // 返回除数为0的错误
                return make_pair(ERROR_DIVISION_BY_ZERO, "Division by zero");
            }
            return make_pair(operand_b / operand_a, "");
        case '^': {
            int result = 1;
            for (int i = 0; i < operand_a; i++) {
                result *= operand_b;
            }
            return make_pair(result, "");
        }
        default: // 返回未知运算符的错误
            return make_pair(ERROR_UNKNOWN_OPERATOR, "Unknown operator");
    }
}

int calculation(const char* input) {
    lock_guard<mutex> lock(stack_mutex);

    // 输入验证
    if (!stack_num || !stack_sym) {
        return ERROR_STACK_NOT_INITIALIZED;
    }

    if (!input) {
        return ERROR_EMPTY_INPUT;
    }

    // 预处理输入，去除空格
    string expression = input;
    expression.erase(remove_if(expression.begin(), expression.end(), ::isspace),
expression.end());
    // 判断非法字符

```

```
for(int i = 0; i < expression.length(); i++) {
    if(expression[i] < '0' && expression[i] > '9' &&
        expression[i] != '+' && expression[i] != '-' &&
        expression[i] != '*' && expression[i] != '/' &&
        expression[i] != '^' && expression[i] != '(' &&
        expression[i] != ')' && expression[i] != '|') {
        return ERROR_INVALID_EXPRESSION;
    }
}

reset_calculation_state();

// 处理表达式
for (size_t i = 0; i < expression.length(); i++) {
    char current_char = expression[i];
    // 处理数字
    if (is_digit(current_char)) {
        int end_pos;
        int number = parse_number(expression, i, end_pos);
        stack_num->push(number);
        i = end_pos - 1;
        continue;
    }

    // 处理操作符
    bool matched = false;
    while (!stack_sym->empty() && should_operator_execute(stack_sym->top(),
current_char)) { //判断优先级
        char top_symbol = stack_sym->top();
        // 判断之前是否有绝对值符号
        if (!abs_cnt && current_char == '|') {
            abs_cnt = 1;
            break;
        }
        // 处理括号匹配
        if (top_symbol == '(' && current_char == ')') {
            stack_sym->pop();
            matched = true;
            break;
        }
        // 检查括号不匹配
        if ((top_symbol == '(' && current_char == '|') ||
            (top_symbol == ')') && current_char == '|') ||
            (top_symbol == '|' && current_char == ')')) {
            return ERROR_PARENTHESIS_MISMATCH;
        }
    }
    // 处理绝对值运算
    if (top_symbol == '|' && current_char == '|') {
        matched = true;
        abs_cnt = 0;
        int value = stack_num->pop();
        if (value < 0) value = -value;
        stack_num->push(value);
        stack_sym->pop();
    }
}
```

```
        break;
    }

    // 执行数学运算
    if (stack_num->size() < 2) {
        return ERROR_INVALID_EXPRESSION;
        // 返回表达式非法的错误
    }

    int operand_a = stack_num->pop();
    int operand_b = stack_num->pop();
    auto calc_result = perform_calculation(top_symbol, operand_a,
operand_b);
    // 除数为0或者表达式非法
    if (!calc_result.second.empty()) {
        return calc_result.first;
    }

    stack_num->push(calc_result.first);
    stack_sym->pop();
}

// 如果没有匹配，将当前操作符入栈
if (!matched) {
    if (current_char == '|' && !abs_cnt) {
        abs_cnt = 1;
    }
    stack_sym->push(current_char);
}
}

// 完成剩余计算
while (!stack_sym->empty()) {
    if (stack_num->size() < 2) {
        return ERROR_INVALID_EXPRESSION;
        // 返回表达式非法的错误 (e.g 1++2)
    }

    char top_symbol = stack_sym->top();
    int operand_a = stack_num->pop();
    int operand_b = stack_num->pop();

    auto calc_result = perform_calculation(top_symbol, operand_a, operand_b);
    if (!calc_result.second.empty()) {
        output_operation_info();
        return calc_result.first;
    }

    stack_num->push(calc_result.first);
    stack_sym->pop();
}

// 检查最终结果
if (stack_num->empty()) {
```

```

        return ERROR_NO_RESULT;
    }

    output_operation_info();
    return stack_num->pop();
}

```

说明：该算法使用两个栈（数字栈和运算符栈）实现中缀表达式的计算，支持运算符优先级和括号处理，且判断了所有表达式非法的情况。为方便理解，所示代码于源代码略有出入，删去了出入栈的记录操作，该操作用于出入栈动画制作。

3.2 多项式类实现 (C++)

```

// polynomial.cpp - 多项式类函数实现
// 额外定义了项类Term，多项式类内部用Term的顺序表管理每一项，实现机制类似stack.hpp，动态
// 扩容，

// 按照项的指数冒泡排序
void Polynomial::sort_terms() {
    for (int i = 0; i < cnt_ - 1; ++i) {
        for (int j = 0; j < cnt_ - i - 1; ++j) {
            if (terms_[j].get_exponent() < terms_[j + 1].get_exponent()) {
                Term temp = terms_[j];
                terms_[j] = terms_[j + 1];
                terms_[j + 1] = temp;
            }
        }
    }
}

// 合并同类项
void Polynomial::combine_like_terms() {
    if (cnt_ == 0) return;

    int write_idx = 0;
    for (int read_idx = 1; read_idx < cnt_; ++read_idx) {
        if (terms_[write_idx].get_exponent() == terms_[read_idx].get_exponent()) {
            int new_coeff = terms_[write_idx].get_coefficient() +
terms_[read_idx].get_coefficient();
            terms_[write_idx].set_coefficient(new_coeff);
        } else {
            ++write_idx;
            terms_[write_idx] = terms_[read_idx];
        }
    }
    cnt_ = write_idx + 1;
}

// 移除系数为零的项
void Polynomial::remove_zero_terms() {
    int write_idx = 0;

```

```
        for (int read_idx = 0; read_idx < cnt_; ++read_idx) {
            if (terms_[read_idx].get_coefficient() != 0) {
                terms_[write_idx] = terms_[read_idx];
                ++write_idx;
            }
        }
        cnt_ = write_idx;
    }

// 添加项
void Polynomial::add_term(const Term& term) {
    resize_if_needed();
    terms_[cnt_] = term;
    ++cnt_;
    sort_terms();
    combine_like_terms();
    remove_zero_terms();
}

// 获取指定索引的项
const Term& Polynomial::get_term(int index) const {
    if (index < 0 || index >= cnt_) {
        throw out_of_range("Term index out of range");
    }
    return terms_[index];
}

// 多项式加法
Polynomial Polynomial::operator+(const Polynomial& other) const {
    Polynomial result(cnt_ + other.cnt_);

    for (int i = 0; i < cnt_; ++i) {
        result.add_term(terms_[i]);
    }
    for (int i = 0; i < other.cnt_; ++i) {
        result.add_term(other.terms_[i]);
    }

    return result;
}

// 多项式减法
Polynomial Polynomial::operator-(const Polynomial& other) const {
    Polynomial result(cnt_ + other.cnt_);

    for (int i = 0; i < cnt_; ++i) {
        result.add_term(terms_[i]);
    }
    for (int i = 0; i < other.cnt_; ++i) {
        result.add_term(Term(-other.terms_[i].get_coefficient(),
other.terms_[i].get_exponent()));
    }

    return result;
}
```

```

}

// 多项式乘法
Polynomial Polynomial::operator*(const Polynomial& other) const {
    Polynomial result(cnt_ * other.cnt_ + 10);

    for (int i = 0; i < cnt_; ++i) {
        for (int j = 0; j < other.cnt_; ++j) {
            int new_coeff = terms_[i].get_coefficient() *
other.terms_[j].get_coefficient();
            int new_exp = terms_[i].get_exponent() +
other.terms_[j].get_exponent();
            result.add_term(Term(new_coeff, new_exp));
        }
    }

    return result;
}

//...

```

说明：该多项式类支持各种运算和字符串转换，可转换成 latex 公式，配合多项式管理类`PolynomialManager`可以实现多个多项式的储存，计算。`PolynomialManager`利用哈希表`unordered_map`管理每个名称对应的多项式。可以基于此原理，设计不仅仅局限于数字，而且还包括字母变量的算数表达式的计算。

3.3 栈动画组件 (React/TypeScript)

```

// StackAnimation.tsx - 栈操作可视化组件
//以下示例完整的动画显示链

<button onClick={handleStepForward} disabled={currentStep >= operations.length}>
    下一步
</button>
// 点击button“下一步”，触发handleStepForward函数

const handleStepForward = () => {
    if (currentStep < operations.length) {
        setCurrentStep(prev => prev + 1);
    }
};
// 设置CurrentStep状态
// React 检测到 currentStep 状态变化，触发：

useEffect(() => {
    const newState = calculateStackState(currentStep);
    setStackState(newState);

    // 设置动画效果
    if (currentStep > 0 && currentStep <= operations.length) {
        const currentOp = operations[currentStep - 1];

```

```

    setAnimatingStack({ type: currentOp.stack_type, index: currentStep });

    // 清除动画效果
    setTimeout(() => {
      setAnimatingStack(null);
    }, 500);
  }
}, [currentStep, operations]);

//设置AnimatingStack, StackState

<div className="stack-container">
  <h4>数字栈</h4>
  <div className="stack">
    {stackState.num_stack.length === 0 ? (
      <div className="stack-item empty">空</div>
    ) : (
      stackState.num_stack.map((value, index) => (
        <div
          key={index}
          className={`stack-item num-item ${
            animatingStack?.type === 'num' && index ===
stackState.num_stack.length - 1
              ? 'animating'
              : ''
            }`}
        >
          {value}
        </div>
      ))
    )}
  </div>
</div>
//初始状态stackState = { num_stack: [], sym_stack: [] }
//渲染结果 <div className="stack-item empty">空</div>
//设置StackState, 重新渲染结果 <div className="stack-item num-item animating">
(e.g.5)</div>

```

说明：该组件通过状态管理和动画效果，实时展示栈操作过程中的数据变化，帮助用户理解算法执行过程。

3.4 Tauri 命令接口 (Rust)

```

// lib.rs - Rust FFI 接口

// 声明外部 C++ 栈函数
extern "C" {
  fn init_stack(capacity: i32) -> i32;
  fn calculation(input: *const std::os::raw::c_char) -> i32;
  fn get_num_operations_count() -> i32;
  fn get_sym_operations_count() -> i32;
  fn get_num_operation_at(index: i32, op_type: *mut i32, value: *mut i32,

```



```
timestamp: *mut i32);
    fn get_sym_operation_at(index: i32, op_type: *mut i32, symbol: *mut i32,
timestamp: *mut i32);
}
// 安全的 C++ 栈函数包装器
fn init_stack_safe(capacity: i32) -> Result<String, String> {
    unsafe {
        let result = init_stack(capacity);
        if result == 0 {
            Ok("Stack initialized successfully".to_string())
        } else {
            Err("Failed to initialize stack".to_string())
        }
    }
}

// Tauri 命令：初始化栈
#[tauri::command]
fn init_stack_command(capacity: i32) -> Result<String, String> {
    init_stack_safe(capacity)
}
```

说明：Rust 代码作为安全的 FFI 桥梁，将 C++ 函数包装成 Tauri 命令，处理错误映射并提供类型安全的接口。总体接口逻辑如下：c++函数处理运算逻辑，将对应的函数接口暴露(extern C)给rust，rust利用这些外部接口(unsafe)，重新保证成另一层安全的接口(safe)，再处理成可以直接和前端进行交互的接口(#[tauri::command])，共三层接口桥梁。

4. 程序运行方式简要说明

4.1 环境要求

- **Node.js:** 推荐 18.x 或更高版本
- **Rust:** 最新稳定版本
- **C++ 编译器:** 支持 C++11 或更高标准
- **包管理器:** pnpm

4.2 安装和运行步骤

步骤 1：安装依赖

```
# 克隆项目
git clone <项目地址>
cd cpp_calculator

# 安装 Node.js 依赖
pnpm install
```

步骤 2：开发模式运行

```
# 启动完整的开发服务器 ( 前端 + Tauri + rust + C++ 编译 )
pnpm tauri dev

# 或仅前端开发 ( UI 修改时使用 )
pnpm dev
```

步骤 3 : 构建生产版本

```
# 构建前端 TypeScript
pnpm build

# 构建完整的 Tauri 应用程序 ( 包含 C++ 集成 )
pnpm tauri build
```

4.3 使用说明

表达式计算器

1. 在输入框中输入数学表达式 (如 : $3+5*2-8/4$)
2. 支持的运算符 : $+$ 、 $-$ 、 $*$ 、 $/$ 、 $^$ (乘方) 、 $| |$ (绝对值)
3. 支持括号改变运算优先级
4. 点击"计算"按钮查看结果
5. 点击"播放动画"查看栈操作过程

多项式计算器

1. 创建多项式 : 输入多项式表达式 (如 : $2x^2+3x-1$)
2. 选择存储位置 (a-e)
3. 进行多项式运算 : 加法、减法、乘法
4. 求导 : 对指定多项式求导数
5. 求值 : 在指定 x 值处计算多项式值

4.4 项目结构

```
cpp_calculator/
├── src/                                # React 前端代码
│   ├── App.tsx                        # 主应用组件
│   ├── StackAnimation.tsx            # 栈动画组件
│   └── ...
├── src-tauri/                          # Rust 后端代码
│   ├── src/lib.rs                    # Tauri 命令定义
│   ├── build.rs                      # C++ 编译配置
│   └── cpp/                          # C++ 计算引擎
│       ├── calc_expression.cpp
│       ├── polynomial.cpp
│       └── stack.hpp
```

```
|— package.json      # Node.js 项目配置
|— tauri.conf.json   # Tauri 应用配置
└— 项目报告.md       # 本报告文件
```

3.5 常见问题解决

- **编译错误**：确保安装了合适的 C++ 编译器 (Windows 上推荐使用 MSVC)
- **端口占用**：开发服务器默认使用 1420 端口，如有冲突可修改配置
- **依赖问题**：删除 `node_modules` 和 `pnpm-lock.yaml` 后重新安装依赖

本程序通过现代化的技术栈实现了高性能的数学计算功能，既展示了算法原理，又提供了良好的用户体验。